FICHA 2
RECURSIVIDADE

2.1 Objetivos

Objetivos que o aluno é suposto atingir com esta ficha:
- Compreender o conceito de recursividade de funções;
- Saber aplicar de forma correta esse conceito, tendo nomeadamente muito cuidado com a escolha das condições de paragem.
- Adquirir alguma sensibilidade sobre quais as situações em que é mais eficaz a aplicação de recursividade.

2.2 Conceito básico da recursividade

Uma função diz-se recursiva quando se chama a si própria, direta ou indiretamente (por intermédio de outras funções). A recursividade permite implementar algoritmos que decompõem o problema a resolver numa sucessão de problemas cada vez mais simples com a mesma estrutura do problema original.

Como há o risco de uma função recursiva poder chamar-se a si mesma um número infinito de vezes, é essencial que exista uma condição que determine o fim do processo recursivo, caso contrário ocorrerá um erro do tipo *stack overflow*. Esta condição determina o momento em que a função deverá parar de se chamar a si mesma. Para se conseguir implementar uma solução recursiva, estrutura-se normalmente a função correspondente em duas partes fundamentais:

1. **Caso Elementar, Caso Base ou Condição de Paragem**. O caso elementar é resolvido sem utilização de recursividade, sendo este ponto geralmente um limite superior ou inferior da regra geral. Dado que não existe a necessidade de se chamar uma nova instância da função, diz-se que este caso corresponde ao ponto de paragem da recursividade.

2. **Regra Geral**. A regra geral da recursividade simplifica a resolução do problema através da utilização recursiva de casos sucessivamente mais simples do problema. O processo repete-se até se atingir o caso elementar que determina o ponto de paragem de todo o processo.

É importante entender que quando uma função se chama a si própria, passa a ser executada uma nova cópia da função, o que implica que as suas variáveis locais são independentes das variáveis locais da primeira cópia, e não podem relacionar-se diretamente.

Existem algumas vantagens e desvantagens na utilização de métodos baseados na recursividade. Algumas delas são sumariadas a seguir.

Vantagens:

- A utilização de uma função recursiva pode simplificar a resolução de alguns problemas;
- Pode-se obter um código mais conciso e elegante nessas situações;
- Uma solução recursiva pode, por outro lado, eliminar a necessidade de manter o controlo manual sobre uma série de variáveis normalmente associadas aos métodos alternativos à recursividade.

Desvantagens:

- As funções recursivas são geralmente mais lentas e necessitam de mais memória na sua execução do que as funções iterativas equivalentes, uma vez que são feitas muitas chamadas consecutivas a funções que implicam a alocação de memória e passagem de parâmetros a cada chamada;
- Um erro de implementação pode levar ao esgotamento dos recursos associados à pilha que gere a chamada a funções. Ou seja, caso não seja indicada nenhuma condição de paragem, ou se essa condição for definida de forma errada e nunca for satisfeita, então o processo recursivo não terá fim.

2.3 Exemplo clássico de recursividade

Para ajudar a compreender o funcionamento de uma função recursiva, utiliza-se geralmente o exemplo do cálculo do fatorial de um número inteiro positivo. O fatorial pode ser definido em função do fatorial do número anterior o que torna a sua implementação um caso típico da utilização de métodos recursivos. Para um dado número inteiro positivo *n*, podemos definir o fatorial como:

$$n! = n.(n-1).(n-2)….2 \,.\, 1 \text{ ou seja } n! = n.(n-1)!$$

Esta função pode ser definida recursivamente em função de um caso elementar e de uma forma geral:

$0! = 1$ (por definição, caso mais simples)

$n! = n \,.\, (n - 1)!$ para $n>0$ (formula recursiva)

Uma possível implementação em C++ de uma função que calcula o fatorial de um dado número *n* é apresentada a seguir.

```
long fatorial(int n) {
  if (n == 0)    /* Caso base que define a condição de paragem */
    return 1;
  else
    return ( n * fatorial ( n-1 ) );
}
```

Note que, enquanto *n* não for igual a 0, a função `fatorial()` chama-se a si mesma com um valor cada vez menor até chegar a 0. A condição `(n==0)` representa o critério de paragem.

Há certos algoritmos que são mais eficientes quando implementados de maneira recursiva. Todavia, a utilização da recursividade deve ser evitada sempre que possível. A utilização de métodos recursivos tende

a consumir mais memória e a ser mais lenta. Lembre-se que é alocado espaço na pilha de valores (é ocupada mais memória) cada vez que o computador faz uma chamada a uma função. A utilização de funções recursivas sem controlo pode esgotar rapidamente a memória do computador.

2.4 Cuidados a ter com funções recursivas.

Se num determinado método recursivo não tiver sido definida uma condição de paragem adequada, poderão surgir alguns problemas. Na função definida anteriormente para o cálculo do fatorial, o que aconteceria se por engano fosse passado como parâmetro o valor -1? A função entraria num ciclo recursivo infinito pois nunca mais a condição de paragem seria válida. O mais provável seria o programa originar um erro de falta de memória da pilha que gere as chamadas a funções (*stack overflow*). Para evitar este tipo de problemas, teremos que garantir que a condição de paragem utilizada contempla todas as situações não desejadas. Uma possível solução para o exemplo do cálculo do fatorial seria a reformulação da condição de paragem:

```
long fatorial(int n) {
  if (n > 1)
    return ( n * fatorial ( n-1 ) );
  else
    return 1;
}
```

Nesta situação seria assumido que apesar de não fazer sentido existir o fatorial de números negativos a função devolveria 1 para esses casos. Se este pressuposto não for considerado aceitável então será necessário encontrar outra solução. Tente alterar esta função de forma a que devolva **-1** quando o número introduzido é negativo.

2.5 Exercícios sugeridos

Pretende-se com o conjunto de exercícios apresentados a seguir que o aluno consolide os seus conhecimentos relativos à recursividade. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Recomenda-se vivamente que o aluno tente resolver em casa os exercícios que não for possível resolver durante as aulas práticas.

Problema 2.1 – Fácil

Escreva uma função recursiva `imprimeAte(int n)` que imprima no ecrã os números inteiros de `0` até `n` (por esta ordem, separados por espaços ou outros caracteres).

Problema 2.2 – Fácil

Escreva uma função que permita escrever no ecrã `n` asteriscos na mesma linha, em que `n` é um parâmetro dessa função, sem utilizar nenhum ciclo (nem `for`, nem `while` e muito menos o `do...while`). O protótipo da função é:

```
void escreveAsteriscos(int n);
```

Problema 2.3 – Fácil

Escreva uma função recursiva que, dados um número real **x** e um número inteiro **n**, retorne $\mathbf{x^n}$. Comece por escrever uma definição recursiva da função $\mathbf{x^n}$.

Problema 2.4 – Fácil

Escreva uma função que devolva o número de *Fibonacci* associado a um determinado inteiro positivo. A sequência de *Fibonacci* **a(1)**, **a(2)**, **a(3)**, ..., **a(n)**, é definida da seguinte forma:

$$a(1) = 1,\ a(2) = 1 \text{ e } a(n) = a(n\text{-}1) + a(n\text{-}2), \text{ para todo o número } n > 2$$

A função assim definida devolve a seguinte sequência de valores:

1, 1, 2, 3, 5, 8, 13, 21, ... , para valores de n=1, 2, 3, 4, 5, 6, 7, 8, …

Problema 2.5 – Difícil

Escreva uma função que converta recursivamente um dado inteiro positivo numa *string* de dígitos.

Problema 2.6 – Médio

Escreva uma função recursiva que converta uma *string* de dígitos no número inteiro por ela representado.

Problema 2.7 – Médio

Escreva uma função recursiva que calcule a soma de uma tabela de inteiros. O protótipo da função é:

```
int soma(int tab[ ], int numElems);
```

Problema 2.8 – Médio

Escreva uma função recursiva que devolva o maior número de uma tabela de inteiros. O protótipo da função é:

```
int maior(int tab[ ], int numElems);
```

Problema 2.9 – Difícil

Escreva uma função recursiva que inverta uma tabela de inteiros de comprimento **n**, utilizando apenas uma variável auxiliar.

Problema 2.10 – Difícil

Escreva uma função recursiva que determine se um número inteiro é ou não uma capicua. Um número é uma capicua se a primeira parte do número é uma imagem espelhada da segunda. Dois exemplos de capicuas são os seguintes números: 1234321 e 56788765. O protótipo da função é:

```
bool capicua(int num);
```

Sugestão: Pode ir comparando recursivamente os dois dígitos extremos do número. Para tal, crie uma função auxiliar que permita eliminar esses dois dígitos de um dado número.

Problema 2.11 – Médio

Implemente uma função recursiva que calcule a função matemática F(n) cuja expressão é indicada a seguir. A variável n representa um número inteiro. Para n negativo, a função que vai implementar tem de devolver -1. Implemente também uma função de teste (função `main()`).

$$F(n) = \begin{cases} 10\,, & \text{para } n = 0 \\ 20\,, & \text{para } n = 1 \\ F(n-1) + 2 \times F(n-2), & \text{para } n > 1 \end{cases}$$

Problema 2.12 – Fácil (saiu no teste de frequência de 12/03/2014)

Implemente a função recursiva F(n) indicada a seguir, assumindo que n é inteiro. Para n negativo, a função retorna 0. Implemente também uma função de teste (função `main()`).

$$F(n) = \begin{cases} 1\,, & \text{para } n = 0 \\ 3 \times F(n-1) + 3, & \text{para } n > 0 \end{cases}$$

Problema 2.13 – Médio (saiu no teste de frequência de 25/03/2015)

Escreva a função recursiva `void decimalToOctal(int num)` que imprime no ecrã a conversão em octal (base 8) do número `num` em base decimal (base 10) passado como parâmetro da função.
Nota: Recorde que a conversão consiste em realizar sucessivamente divisões inteiras por 8 do número passado à função. Os restos de cada divisão inteira formam os dígitos do número codificado em octal.
Exemplo: $156_{10}=234_8$, ou seja: 156/8(4)=19 /8(3)=2 /8(2)=0
Os números entre parêntesis correspondem ao resto da divisão.

Problema 2.14 – Fácil (saiu no exame de recurso de 03/07/2017)

Utilizando uma função recursiva, implemente a função indicada a seguir:

$$\begin{cases} T(n \leq 0) = 0 \\ T(1) = 1 \\ T(n \geq 2) = (T(n-1) + T(n-2))/2 \end{cases}$$

Problema 2.15 – Médio (adaptado do teste de frequência de 13/04/2018)

Escreva o código da função recursiva `unsigned int octalParaDecimal(unsigned int num)` que devolve a conversão em decimal (base 10) do número `num` em octal (base 8) passado como parâmetro da função. Assuma que `num` é sempre um número representado corretamente em octal, *i.e.* que só contém dígitos de 0 a 7. Recorde que a conversão consiste em realizar a soma pesada dos dígitos octais do número passado à função, pesados com potências de 8 de expoente dependente da posição dos dígitos. A função não deve escrever nada no ecrã. Implemente também uma função de teste (função `main()`).
Nota: O resto da divisão inteira de `num` por 10 permite extrair o dígito mais à direita. O quociente desta divisão inteira permite obter os restantes dígitos. A conversão recursiva consiste em somar o dígito mais à direita com a conversão do número constituído pelos restantes dígitos (problema mais pequeno) multiplicada por 8.
Exemplo: $273_8 = 187_{10} = 3 * 8^0 + 7 * 8^1 + 2 * 8^2 = (3) + 8 * (7 + 2 * 8^1) = (3) + 8 * [\ (7) + 8 * (2)\ ]$.